Introduction to Modern Fortran

External Names, Make and Linking

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Introduction

Some of this copies parts of the modules lecture
It would be too confusing otherwise

External Procedures (1)

A file can contain more than one module
Also procedures, not included in modules
These are called external procedures

```
SUBROUTINE dongle
    . . .
END SUBROUTINE dongle
SUBROUTINE mangle
    . . .
END SUBROUTINE mangle
```

PROGRAM is always an external procedure

External Procedures (2)

- They are not recommended in most cases

It's much harder to get right, and link correctly

- But a lot of older programs do it

Same mechanism as used to call external libraries
And code written in C or using a C interface
LAPACK, MPI, . . .

External Procedures (3)

This is a little better, and useful for testing

- But note that the order is critical

```
MODULE whatever
      . . .
END MODULE whatever

MODULE whatsit
      USE whatever
      . . .
END MODULE whatsit

PROGRAM mangle
      USE whatsit
      . . .
END PROGRAM mangle
```

Compiling Modules (1)

This is a FAQ – Frequently Asked Question
The problem is the answer isn't simple

- That is why I give some of the advice that I do

The following advice will not always work
OK for most compilers, but not necessarily all

- This is only the Fortran module information

And it doesn't apply to IBM AIX . . .

Compiling Modules (2)

The module name need not be the file name
Doing that is strongly recommended, though

- The same applies to external procedures

You now compile the file, but don't link it

```
nagfor –C=all –c mymod.f90
```

It will create files like mymod.mod and mymod.o
They contain the interface and the code
Procedures create only code files (mymod.o)

Will describe the process in more detail later

Using Compiled Modules

All the program needs is the USE statements

- Compile all of the modules in a dependency order

If A contains USE B, compile B first

- Then add a *.o for every file when linking

```
nagfor –C=all –o main main.f90 mymod.o

nagfor –C=all –o main main.f90  \
      mod_a.o mod_b.o mod_c.o
```

Makefile Warnings

This does NOT teach how to use make
It teaches just the Fortran–specific aspects

See Building, installing and running software
If you haven't been to it, DO SO before starting!

The defaults for $(FC) and $(FFLAGS) are broken
Hopelessly outdated, and no longer work

That applies to both POSIX and GNU make!
$\Rightarrow$ You must set them yourself
Or you can use other names, if you prefer

Makefile Basics (1)

Use make in exactly the same way as for C

- Must set $(FC) and $(FFLAGS) or whatever
- Modules create both *.mod and *.o files
- Do not need to set LDFLAGS = –lm

Will give a very simple example:
The module file utils.f90 creates a module UTILS
And that is used by a program file trivial.f90

- Dependencies include both *.mod and *.o files

Makefile Basics (2)

```
FC = nagfor
FFLAGS = -C=all

LDFLAGS =

all: trivial

utils.mod utils.o: utils.f90
<tab> $(FC) $(FFLAGS) -c utils.f90

trivial: utils.mod utils.o trivial.f90
<tab> $(FC) $(FFLAGS) $(LDFLAGS) -o trivial trivial.f90 utils.o
```

Interfaces in Modules

The module can define just the interface
The procedure code is supplied elsewhere
The interface block comes before CONTAINS

- The best way of calling external procedures

Including external libraries, C code etc.

- You had better get them consistent!

The interface and code are not checked

- Extract interfaces from procedure code

f2f90 can do it automatically

Cholesky Decomposition

```
SUBROUTINE CHOLESKY(A)
      USE double    ! note that this has been added
      INTEGER :: J, N
      REAL(KIND=dp) :: A(:, :), X
      N = UBOUND(A, 1)
      DO J = 1, N
            X = SQRT(A(J, J) -    &
            DOT_PRODUCT(A(J, :J-1), A(J, :J-1)))
            A(J,J) = X
            IF (J < N) &
                  A(J+1:, J) = (A(J+1:, J) -    &
                  MATMUL(A(J+1:, :J-1), A(J, :J-1))) / X
      END DO
END SUBROUTINE CHOLESKY
```

The Interface Module

```
MODULE MYLAPACK
      INTERFACE
            SUBROUTINE CHOLESKY (A)
                  USE double    ! part of the interface
                  IMPLICIT NONE
                  REAL(KIND=dp) :: A(:, :)
            END SUBROUTINE CHOLESKY
      END INTERFACE
! This is where CONTAINS would go if needed
END MODULE MYLAPACK
```

The Main Program

```
PROGRAM MAIN
      USE double
      USE MYLAPACK
      REAL(KIND=dp) :: A(5,5) = 0.0_dp, Z(5)
      DO N = 1,10
            CALL RANDOM_NUMBER(Z)
            DO I = 1,5 ;    A(:,I) = A(:,I)+Z*Z(I) ;    END DO
      END DO
      CALL CHOLESKY(A)
      DO I = 1,5 ;    A(:I-1,I) = 0.0 ;    END DO
      WRITE (*, '(5(1X,5F10.6/))') A
END PROGRAM MAIN
```

The Makefile

```
FC = nagfor
FFLAGS = -C=all
LDFLAGS =

all: program

cholesky.o: cholesky.f90
<tab> $(FC) $(FFLAGS) -c cholesky.f90

 mylapack.mod mylapack.o: mylapack.f90
<tab> $(FC) $(FFLAGS) -c mylapack.f90

program: cholesky.o mylapack.mod mylapack.o program.f90
<tab> $(FC) $(FFLAGS) $(LDFLAGS) -o program.f90    \
<tab> cholesky.o mylapack.o
```

External Names (1)

The following names are global identifiers

All module names
All external procedure names
Old Fortran COMMON blocks

- They must all be distinct

And remember their case is not significant

- Avoid using any built–in procedure names

That works, but it is too easy to make errors

External Names (2)

C and C interfaces add more:

Some BIND(C) names (see later)

C file scope extern declarations

Almost all C library functions

- Also many C programmers are sloppy

Undocumented external names are common mistakes

Few people have trouble with pure Fortran code

Build Warnings

Avoid file names like fred.f90 AND
external names like FRED
Unless FRED is inside fred.f90

- It also helps a lot when hunting for FRED

This has nothing at all to do with Fortran
It is something that implementations get wrong
Especially the fancier sort of debuggers
It applies just as much to C code

More on Makefiles

It's useful to know a bit more about makefiles
The remainder is some of the how they work

What Compilers Do (1)

A file frederick.f90 contains modules fred and alf
You compile this with:

```
nagfor –C=all –c frederick.f90
```

It will create files frederick.o, fred.mod and alf.mod

- frederick.o contains the compiled code

Link this into into the executable, in the usual way:

```
nagfor –C=all program program.f90 frederick.o
```

What Compilers Do (2)

- fred.mod and alf.mod contain the interfaces

Think of them as being a sort of compiled header

- You don't do anything with these, explicitly

The compiler will do find them and use them

A file program.f90 contains USE fred and USE alf

- The compiler will search for fred.mod and alf.mod

Searched for using the same paths as headers

To add another search path, use –I<directory>

- Be warned – compilers vary – see their docs

Makefile Rules (1)

You need to set up rules to compile the modules
And to add dependencies to ensure they are rebuilt

- Dependencies are exactly like headers

The object file has a dependency on the module

A lot of people forget about headers in makefiles

- Doing that with modules is disastrous

Gets the compiled code out of step with the interface
E.g. gets the new fred.o and the old fred.mod

Makefile Rules (2)

A file program.f90 contains USE fred and USE alf
Modules fred and alf are in files fred.f90 and alf.f90
This is how you set up the dependency and rules:

```
program: program.o fred.o alf.o
<tab> $(FC) $(FFLAGS) $(LDFLAGS) -o program

program.o: program.f90 fred.mod alf.mod
<tab> $(FC) $(FFLAGS) -c program.f90

fred.mod fred.o: fred.f90
<tab> $(FC) $(FFLAGS) -c fred.f90

alf.mod alf.o: alf.f90
<tab> $(FC) $(FFLAGS) -c fred.f90
```

Makefile Rules (3)

Say frederick.f90 contains modules fred and alf
and includes the statement USE double

```
program: program.o frederick.o double.o
<tab> $(FC) $(FFLAGS) $(LDFLAGS) -o program

program.o: program.f90 fred.mod alf.mod
<tab> $(FC) $(FFLAGS) -c program.f90

double.mod double.o: double.f90
<tab> $(FC) $(FFLAGS) -c double.f90

fred.mod alf.mod frederick.o: frederick.f90 double.mod
<tab> $(FC) $(FFLAGS) -c double.f90
```

Doing Better (1)

Can clean up the Makefile somewhat, fairly easily

E.g. use the $@, $< and $* macros
But take care, as things are a little tricky

- Problem is one module file produces two results

And headers are not compiled, but modules are

It's still a bit tedious with a lot of modules

Doing Better (2)

You can do a good deal better, but it's advanced use
Beyond Building, installing and running software

Need either inference rules or pattern rules
Worse, POSIX and GNU are wildly different

It can be done, and it's not even very difficult

- But it is very system–dependent!